DEAD: Dead Code Elimination based Automatic Differential Testing

Master Thesis

Author(s):
Girsberger, Yann W.

Publication date:
2022-04-01

Permanent link:
https://doi.org/10.3929/ethz-b-000547786

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DEAD: Dead Code Elimination based Automatic Differential Testing

Master's Thesis

by

Yann W. Girsberger

Advanced Software Technologies Lab
ETH Zürich

Supervised by

Theodoros Theodoridis
Prof. Dr. Zhendong Su

April 1, 2022

ABSTRACT

In their pursuit of faster code, compilers invoke increasingly more complex optimizations. Along with the increase in complexity the difficulty to identify missed optimizations and performance regressions also increases. Most existing work does not focus on finding such missed opportunities. It either concentrates on correctness, does not pinpoint the precise missed optimization, or only finds missed optimizations for specific components of a compiler.

We built DEAD, a tool that leverages Dead Code Elimination and Differential Testing to find regressions, and generate corresponding bug reports. We conducted an empirical study on the 647 regressions found by DEAD and developed a new analysis to assess the evolution of LLVM and GCC. With DEAD, we help to improve the quality of compilers and enable further thorough testing of new changes.

Contents

1 INTRODUCTION

Compilers need to work at constantly changing frontiers. The User-Machine-Frontier, where a compiler has to explain in a user-friendly way, what the user has done wrong. On the Machine-Machine-Frontier, where a compiler has to understand another machine and translate user-given commands into machine commands. Often times, it is not only for a single type of machine, but many different ones. Additionally, the resulting product is required to be as performant as possible on both frontiers.

As the scope of a compiler's task is of such breadth, compilers themselves become very complex programs, written by many people and containing many non-trivial interactions with themselves. Thus, changing one component of a compiler might lead to unintended side effects for other components, cause loss of performance, or even wrong results in non-obvious cases. Finding and reporting such cases help to fix errors, prevent performance regressions and lead to a deeper understanding of the affected component. Unfortunately, non-obvious cases are difficult to find.

Dead Code Elimination is a component of a compiler which searches for code segments that are never executed and thus of no importance to the resulting program. However, before eliminating a code segment the compiler must prove that it is never executed. This is a fragile undertaking and sensitive to changes to the compiler. [Theodoridis et al. 2022] leverages this sensitivity of Dead Code Elimination to find missed optimizations across compilers.

DEAD We extend the work of [Theodoridis et al. 2022] by developing DEAD, a tool for finding compiler regressions and generating bug reports. So far, DEAD found 647 regressions across LLVM and GCC. In an empirical study, we:

- analyzed the relative frequency of the 647 regressions and found that most of the missed optimizations found by DEAD yield from only a few regressions.
- found that most regressions discovered by DEAD are related to Peephole Optimizations and Value Range Analysis.
- were able to derive the structure of the optimization pipeline of the compiler from the regressions.

Furthermore, we propose a novel analysis that allows us to study the evolution of compilers w.r.t dead code elimination. We show this analysis by comparing GCC and LLVM, which demonstrates that:

- both GCC and LLVM improve their ability to eliminate code while introducing comparatively few regressions.
- LLVM has a significant advantage over GCC.

Additionally, we made DEAD available online at github.com/DeadCodeProductions/dead which includes a continuously updated list of regressions that have been reported.

Overall, we show the effectiveness of DEAD in its main use-case of finding regressions and highlight several others.

1.1 Related Work

In the last decade, the amount of research done in automatic compiler testing has increased significantly according to [Chen et al. 2020]. However, most of these works focused on finding correctness bugs, and little work was done on finding performance regressions.

Typically, performance regressions are detected via benchmarks such as SPEC 2017 [Bucek et al. 2018]. However, as regressions may manifest across multiple changes to the compiler, performance benchmarks are not suitable for pinpointing the exact changes leading to a regression. DEAD does not exhibit this limitation and returns the exact commit for each regression found.

[Barany 2018] is the most similar work to this thesis. It uses the same general procedure for finding bugs. However, to detect a bug, they rely on comparing the produced assembly directly via different specialized checkers that e.g. count the number of memory accesses, SIMD instructions, floating-point instructions, etc. In comparison, DEAD relies on the results of [Theodoridis et al. 2022] that leverages dead code elimination to find regressions. Additionally to the reduction of the bug, DEAD also bisects it to ensure its uniqueness. Lastly, DEAD has been open-sourced[1].

[Taneja et al. 2020] does maximally precise data-flow analysis to find unsoundness in the transformations for LLVM. According to them, no unsound transformations were found but they were able to improve the precision of several data-flow analyses, allowing for better transformations and thus higher performance. DEAD does not specifically target such analysis but any data-flow-related optimizations.

Missed Optimizations [Hashimoto and Ishiura 2016] discover missed optimizations of arithmetic expressions. Two equivalent arithmetic expressions are generated by removing dead code form the initially generated expression in AST-form. The assembly of the original and the optimized expression are then compared to identify missed optimizations.

[Gong et al. 2018] studies the stability of loop optimizations. They extract loops from benchmarks and search for each extracted loop the best performing semantically equivalent loop. With the results gained when searching for the best loop, they assess the strengths and weaknesses of the loop optimizations of the compiler, potentially revealing missed optimizations. DEADs approach is more generic than both [Hashimoto and Ishiura 2016] and [Gong et al. 2018], which target specific components of the compiler.

Automatic Correctness Testing [Le et al. 2013, 2015] use the concept of Equivalence Modulo Inputs to find miscompilations. They derive equivalent programs by first detecting segments of dead code in an executable and then altering these segments. Such a change should not influence the output of the initial program. This approach can not be used to find performance bugs.

CSmith by [Yang et al. 2011] is one of the fuzzing projects starting the recent interest in automatic compiler testing. CSmith generates executable C programs which are free of undefined behavior, to stress-test all compilers processing C code. It is heavily used in many automatic compiler testing projects, also in DEAD. However, it focuses on finding compiler errors and not missed optimizations.

Yarpgen by [Regehr et al. 2012] is another C code generator that follows CSmith. It focuses on finding miscompilation bugs by comparing the output of the same generated program compiled by different compilers.

RandIR by [Ofenbeck et al. 2016] finds bugs in transpilers of DSLs. They generate a random AST from the grammar of the DSL and instantiate it in the DSL and a known good language (Scala). They compare the output of the DSL and the known good program to find bugs in the transpiler of the DSL.

[1] github.com/DeadCodeProductions/dead

2 BACKGROUND

In this section, we are going to present the different concepts and technologies used in DEAD.

2.1 Differential Testing

Unit-tests are a well-established test mechanism for programs. A single test consists of a procedure to run, its inputs, and the expected outcome. The latter is provided by an oracle, which is usually the programmer. However, sometimes the solution is unknown, infeasibly expensive to get, or not unique, i.e., there is no feasible oracle for the unit test. To still be able to test a program p without an oracle, differential testing uses multiple different implementations that solve the same problem as p and compares their behavior on the same input. We expect all implementations P to have the same behavior/output on a given input i, as they solve the same problem. If our expectation does not hold, we found a (potentially) unwanted behavior. If our expectations hold, we can continue in one of two ways:

- **unguided differential testing:** Get a new i' for which we expect all $p \in P$ to produce the same behavior and try again.
- **guided differential testing:** Analyse the behavior of all $p \in P$ and inform the generation of the new input i' with the results. Try again with i'.

Guided differential testing can reduce the number of inputs that need to be generated to find an unwanted case compared to unguided differential testing but requires an understanding on how to craft a better input from running the program. Unguided differential testing does not require such an understanding and can thus treat the programs P as a black box. This is a desirable property when the programs are not closely comparable or analysis of them is prohibitively expensive. Differential testing for testing software has been in use for more than 20 years. [McKeeman 1998] patented using differential testing for testing compilers in 1998.

2.2 Finding Regressions via Dead Code Elimination

When applying differential testing to compilers, natural instantiations of programs P are the different *versions* of a compiler. The identical input for all programs is then any source code the compiler understands. To improve the efficiency of the compiled code, compilers optimize the given source code. Depending on the optimizations applied, the resulting assembly of the input may be different. A compiler might not be able to perform a given optimization, although it would have been possible. Such an optimization has been *missed* by the compiler. [Theodoridis et al. 2022] combined differential testing with Dead Code Elimination (DCE) to find such missed optimizations.

```
1 int main(){
2     if (0)
3         printf("Dead World!");
4     int i = 0; // Dead
5     printf("Hello World!");
6 }
```

Listing 1. Example of Dead Code. The print statement on line 3 will never be executed as if(0) always evaluates to true. Similarly, integer i is dead as it does not effect the print statement on line 5, which is the only statement with a side effect in this example.

```
1 static void f(int arg){
2     for (int j=237; j<arg; j++){
3         printf("All work and no play");
4     }
5 }
6 int main(){ f(237); }
```

Listing 2. More complicated example of Dead Code. The whole program has no side effect. The call to f on line 5 will return without printing "All work and no play" as j equals arg.

Dead code is a part of a code that does not influence the side effects of the program,i.e., can not influence the result/output of the program. Listing 1 shows two simple examples of dead code: the `if`-condition is never true and `i` is never used and not accessible from anywhere else, thus the print statement and `i` can be removed safely. In Listing 2 we see a more difficult example of dead code. The call to function `f` can be removed as the condition `j < arg` is never true. Furthermore, the entirety of `f` can be removed, as it is not used anywhere else in the code. Dead Code *Elimination* is an optimization of the compiler that identifies dead code and removes it, i.e., *eliminates* it. Elimination is strongly dependent on good code analysis and compiler heuristics. In Listing 2, if the compiler does not check the argument passed to the function and notices that there is no other call to `f`, the dead code will not be detected and remains part of the program.

A priori it is unclear which optimizations, including DCE, can be applied to a given code. Even when given the code and the compiled assembly, it is difficult to find which part of the code DCE has been applied to as we have to test for the absence of code that may have been merged with other parts via common subexpression elimination or inlining.

To deal with this, we put markers in each if-statement and for-loop of the program. A marker is a call to a function that takes no arguments and is only declared, i.e., has no implementation. Compilers are only allowed to remove such calls *iff* they can prove that they are dead code. Additionally, no inlining of the markers can occur as the function bodies are not known to the compiler. We call the process of putting markers into the basic blocks of if-statements and for-loops **instrumenting**.

```c
void marker_0(void);
static void f(int arg){
    for (int j=237; j<arg; j++){
        marker_0();
        printf("All work and no play");
    }}
int main(){ f(237); }
```

Listing 3. Instrumented code of Listing 2

Inspecting the assembly output of instrumented code will show which markers were *not* eliminated (are alive). Suppose now we have two versions A and B of a compiler with B being the newer version. After compiling Listing 3 and inspecting the assembly for A and B, we find that compiler A was able to eliminate `marker_0`, i.e., it was not found in the assembly, while B included the marker and thus was not able to eliminate it. We call such a behavior a *compiler regression.*

We note that this work assumes both A and B to be correct w.r.t. the tested inputs.

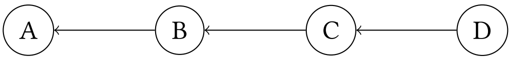
2.3 Git

Git is a popular version control system in the software industry. Its main use is to track changes to the source code of a program. This allows a developer to pinpoint the exact moment when a change was introduced to the code. Such changes applied to the program are called *commits*. Through explanations given with every commit, called commit messages, a person in the future can find out what motivated the changes.

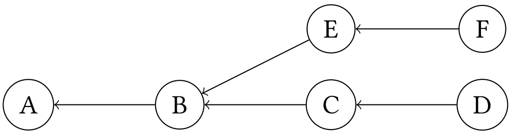
Git History: The git history is (in theory) the complete list of changes ever made to the source code, from the first time the files have been created up to their current state. The simplest non-trivial git history is a linear history. Each commit comes after the other, resulting in a directed path. The direction is given by the parent relation:

Definition 2.1. **Parent Relation**: A commit *A* that comes *directly before* a commit *B* is called the **parent** of B.

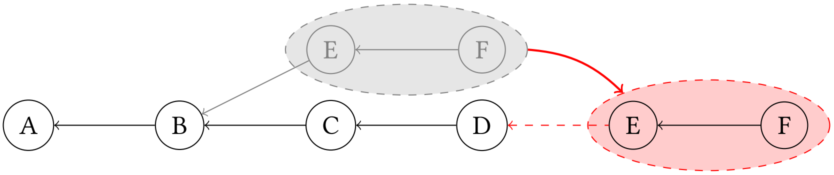
Definition 2.2. **Ancestor Relation**: The transitive closure of the parent relation constitutes the **ancestor** relation.



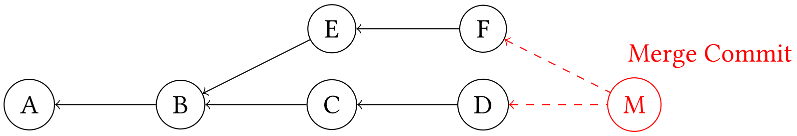
As some developers may not work together at the same time, git allows a user to *branch* off the linear history, continuing their own, locally linear history. The branch that is considered the current state of the project, is called the `main` or `master` branch. As both branches continue to accumulate changes, the git history now becomes a tree.



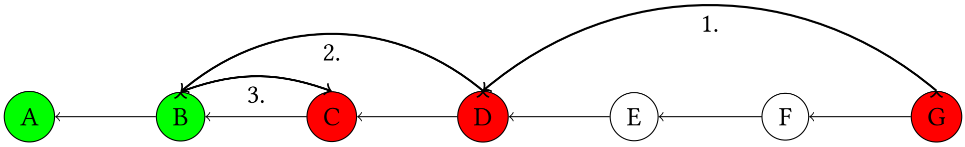
Applying the changes of one branch to another can be achieved in different ways. The two ways highlighted here are *merging* and *rebasing*. Rebasing takes all of the changes made as they appear in the branch and tries to apply them to the last commit of the other branch one by one. We are choosing a new base to start from. If successful, the resulting history w.r.t. the branch that has been rebased and the branch that has been rebased upon is now a path again.



Merging on the other hand takes the top of both branches and creates a new entry in the history, combining both sets of changes. The history is no longer a tree.



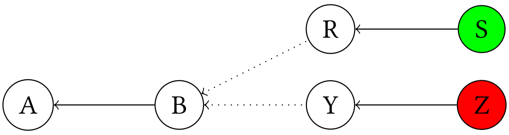
Bisection When encountering an error in the project that is tracked in git, the git history allows to backtrack in which entry the error first appeared. Given a known "good" entry, i.e., where the error does not appear, and a known bad entry as well as a way to check for the error, the bug-introducing commit can be found by binary searching on the path between A and G. First, D is tested as it is midway between A and G. D turns out to be bad, thus the range gets updated to A to D. The next midpoint is (rounded down) B, which is good, shrinking the range to B to D. C, the last entry to be checked, yields a bad result on the test. We can now see that C is the first entry that exhibits the bad behavior.



When the good and bad entry are on different branches as depicted below, we first test the entry where the branch branched off `main`. This is formally defined as

Definition 2.3. **Best Common Ancestor:** The best common ancestor (BCA) of two entries E and E' in a git history is the entry A which is the ancestor of both E and E' and there is no A'≠A which is ancestor of E and E' and A is ancestor of A'.

In the diagram below, B is the best common ancestor of S and Z.



When B is good we have to bisect between B and Z which is a normal bisection situation and we can apply the algorithm discussed previously. When B is bad, we negate the test and apply the bisection algorithm on the range B to S. In this case, we do not find the entry that first broke the project but the one that fixed it. To also find an entry that introduced the problem, we test the oldest entry we want to test if it exhibits the error or not. In our example, this is A. If A is good, we can now start a bisection between A and B (as we already know that B is bad). If A is also bad, we have to abort.
When running DEAD, we have encountered the case that B is bad, however infrequently enough that the additional implementation effort to handle the case in all other parts of DEAD outweighed the benefits of handling regressions that are caused by a beneficial addition on a release branch [2].

It is important to note that LLVM and the parts of GCC we analyzed include practically no merge commits, discarding the need for a bisection algorithm for merges.

2.4 CSmith

CSmith is a random program generator for C and was developed by [Yang et al. 2011]. Programs generated by CSmith are free of undefined behavior. It is the only source of input programs for DEAD and at the core of the generator subsection 3.1.

2.5 CReduce

CReduce is a program to derive a small piece of code exhibiting a specific behavior from a larger piece of code exhibiting the same behavior, i.e., the larger piece of code is *reduced* to the smaller one. To be able to reduce code, CReduce requires a test for the wanted behavior and the initial code exhibiting the behavior.
It was developed by [Regehr et al. 2012] and is extensively used in the reducer of DEAD (subsection 3.4).

[2] They are in a sense "positive regressions". The commits found when bisecting such a case allow for more eliminations.

3 DEAD

Finding and fixing bugs is hard, especially in complex systems such as compilers. We aim to create a tool that finds regressions and other missed optimizations in compilers and generates actionable bug reports for developers automatically.

From the problem statement above we can directly split DEADs functionalities into two parts: Finding a regression and generating a corresponding bug report.

Finding a regression DEAD uses unguided differential testing (subsection 2.1) to find regressions. Therefore DEAD requires something which generates a new candidate input and a testing functionality to determine if the candidate input is a regression. Based on their functionality we name these parts Generator (subsection 3.1) and Checker (subsection 3.2) respectively.

Report For a report to be actionable, it should:

- include a problem description.
- be reproducible.
- include a small/minimal code example that triggers the bug.
- include since when the bug is occurring.
- include further processing of the code (Intermediate Representation, Assembly) as needed or wished for.

From these requirements, we can directly derive two further parts of the structure of DEAD, namely one that reduces the code of found bugs to a small example and another part searches when the bug was introduced, named Reducer (subsection 3.4) and Bisector (subsection 3.3) respectively.

With these components, we can define the high-level flow chart of DEADs (mode of) operation, depicted in Figure 1.

Definition 3.1. **Compiler Setting:** A compiler setting is a specific compiler with a specific optimization level enabled.

The inputs of DEAD are two sets of compiler settings: the targets and the attackers. The targets are the settings that ought to eliminate every marker that the attackers can eliminate. Typically, the targets are the trunk of the compiler project with all its possible optimization levels and the attackers are any number of the previous releases, also with all optimization levels. DEAD will start to generate candidates which are checked for exhibiting a regression w.r.t. the given targets and attackers. We will refer to a candidate that does exhibit a regression as an “interesting” candidate.

Definition 3.2. **Interestingness:** A candidate is called *interesting* in the context of targets and attackers if there exists a marker in the candidate that at least one compiler setting in the attackers can eliminate and at least one compiler setting in the targets can not.

If an interesting candidate is found we can bisect the point in time where the regression was introduced. This is of course only possible if the optimization level of the attacker and the target that results in an interesting case are identical. The candidate is then reduced to a small example used to generate the final report.

3.1 Generator

The goal of the generator is to return an instrumented candidate. We first generate a piece of runnable code with CSmith (subsection 2.4) and then instrument it as described in subsection 2.2. CSmith allows to enable or disable certain constructs of the C language such as unions, structs, arrays, pointers, and more. To prevent encountering undefined behavior, we always disable unions, volatile fields, and volatile pointers, and enable safe math. All other options are enabled or disabled at random when generating a new file.

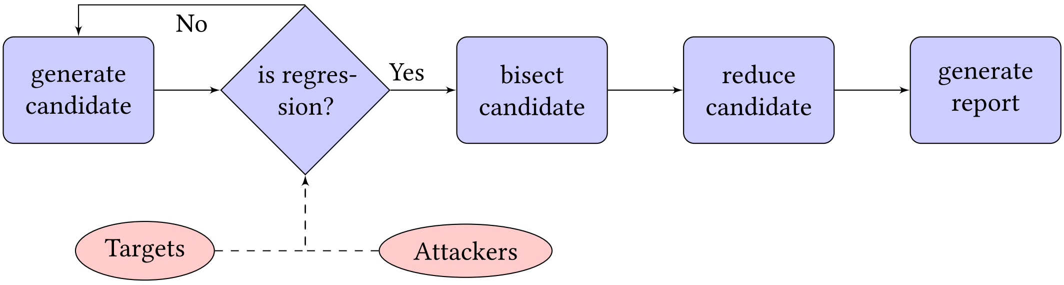


Fig. 1. High-level flow chart of DEAD.

Furthermore, we restrict the length of the program (including whitespaces) to be at least 20'000 characters and at most 50'000 characters. Candidates of great length (500'000 - 1'000'000 characters) have a higher chance of containing a regression as there are a lot more markers and thus opportunities, however reducing such a candidate takes multiple hours on average while a candidate of smaller size can be reduced in under an hour. Manual testing showed that files of smaller size than 20'000 characters are rarely interesting, resulting in a long search time for interesting candidates that outweighs the benefits of the reduced reduction time.

3.2 Checker

The checker has two similar modes of operation.

- **Find**: Given a candidate, targets, and attackers, check the candidate's sanity, i.e., if it exhibits any undefined behavior, and interestingness.
- **Re-assert**: Given a *modified* interesting candidate, the interesting marker of that candidate, the target, and the attackers exhibiting the regression, check the sanity of the modified candidate as well as its interestingness.

"Find" is used to test new candidates produced by the generator. It corresponds to the decision node depicted in Figure 1. "Re-assert" is the test used in the reducer to verify that the modification performed on the interesting candidate still results in a sane interesting candidate. The sanity and interestingness check each consist of multiple parts. Each part must pass for the check to pass.

The sanity of a candidate is tested as follows

- All markers get an empty function body, essentially removing them from the file. Then the candidate with the empty marker bodies is compiled with the address and undefined behavior sanitizers enabled. A known-good compiler is used, which is usually the one provided by the underlying OS. The sanitizers require the code to be runnable, a property that is provided by CSmith.
- The candidate is compiled and the output is checked for warnings that indicate a (potentially) ill-defined program. Checked warnings include obvious problems such as `division by zero`, type mismatches or problematic conversions such as `comparison of distinct pointer types` and `pointer from integer`, and structural problems such as that the code is invalid in `C99`, the main function not returning an integer or a non-void function returning void.
- The candidate is compiled with CompCert which is a formally verified optimizing compiler which understands most of ISO C 2011 [Com 2022].

The interestingness check differs for the two modes. When given a candidate from the generator, the interestingness is unclear, thus the candidate is compiled for all targets and attackers, and for each the markers that are alive are extracted as described in subsection 2.2. The sets of markers of the targets are then compared to the attackers' set; if any of the targets' sets include a marker that any of the attackers' sets does not include, a regression has been found. The target set that includes the marker and all attackers which do not include the marker are then returned by the checker.

When invoked by the reducer the checker has to verify the interestingness of a `modified` candidate i.e. it is already known in which way the candidate *should* be interesting, and we want to know whether it still is. The checker extracts alive markers from the target setting and attacker settings and checks if they show the same behavior. If so, the modified candidate is still interesting.

3.3 Bisector

The inputs to the bisector are an interesting candidate with the respective target and potentially multiple attackers. The output of the bisector is the commit that introduces the regression or an error if the commit could not be found due to failing compiler builds. During the bisection, the target commit acts as the "bad" marker while the attacker whose best common ancestor with the target marker is the closest to the target marker, is the "good" marker. This is to not unnecessarily increase the range to bisect over as well as to guarantee a deterministic start. The bisector has now met all criteria to execute the bisection algorithm for a tree-like history described in subsection 2.3.

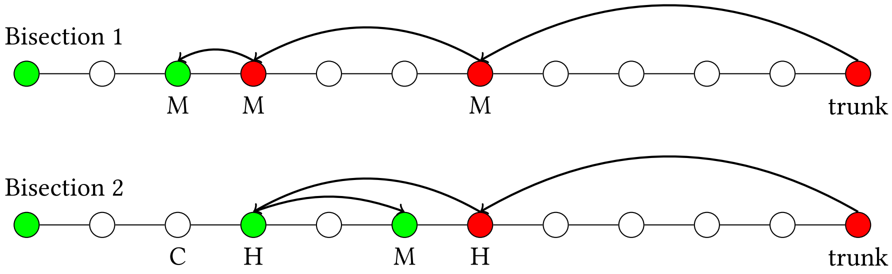


Fig. 2. The figure shows two bisections for different commits over the same range. An "M" or "H" below a node indicates a cache miss or hit respectively. A "C" indicates that the commit is already cached. *Bisection 1* works on a cold cache, thus having a 100% miss rate. In *Bisection 2* we already see the miss rate to 33%, showing the effectiveness of the compiler cache. This effect gets more pronounced the higher populated the cache and the greater the range is.

Optimizing for Compiler Reuse: To check the interestingness throughout the bisection process, the compiler at the encountered commit for each step must be built. Compiling each commit every time anew is prohibitively expensive computationally. The natural solution is to cache all compilers ever built by DEAD. As the attackers consist practically always of the same members, namely the major and minor releases, and the target is always trunk or on `main`, there are few different initial commits when starting the bisection. We call the commits that lie between the good and bad commit the *commit/bisection range.* As minor releases are on the same branch as major releases, we have the same amount of ranges as there are major releases. In Figure 2 we can see two bisections and the evolution of the underlying cache. *Bisection 1* works on a cold cache leading to 100% misses. However, already the second bisection can utilize at least one of the cached compilers. In the instance of *Bisection 2*, only one additional compiler needs to be built. The effectiveness of the cache increases as the bisection range grows as more common "halving steps" are needed for each bisection. The ranges encountered by DEAD include between 7'000 [3] and 160'000 [4] commits, highlighting the importance of an efficient caching and bisection system.

Finding regressions quickly after they have been introduced is important, as the responsible developer may still remember the reasoning around the changes made. Therefore regularly updating trunk is essential. Updating trunk, however, changes the history ranges to bisect over and thus the midpoints which in turn are most likely not in the cache. Furthermore, the chance of the old midpoints being hit during the bisection is small due to their proximity to the new midpoints. In Figure 3 we can see the effects of such an update on the previous example. The history is extended by two commits in the first step. The next bisection, *Bisection 3*, then misses twice because of the shifted midpoints even though *Bisection 3* bisects to the same commit as *Bisection 1* .
To prevent this we implemented a **cache-informed bisection**, which behaves as follows:

(1) Determine the commits of the range to bisect are already cached.
(2) Perform a bisection, but only over the already cached commits. We will refer to this as **in-cache bisection**.
(3) Update the original range with the information gained from the in-cache bisection.
(4) Run a normal bisection on the updated range.

Bisection 3' in Figure 3 shows the cache-informed bisection in the same situation as *Bisection 3*. *3'* is unaffected by the update of trunk and has a 100% hit rate as the regression commit was previously bisected. The cost of cache-informed bisection is that additional tests are made when the commit to be bisected is outside a densely cached region. For example, in Figure 3 the region to the left of trunk is sparsely cached. However, additional tests performed on compilers in the cache are significantly cheaper than one test on a new compiler, outweighing the costs.

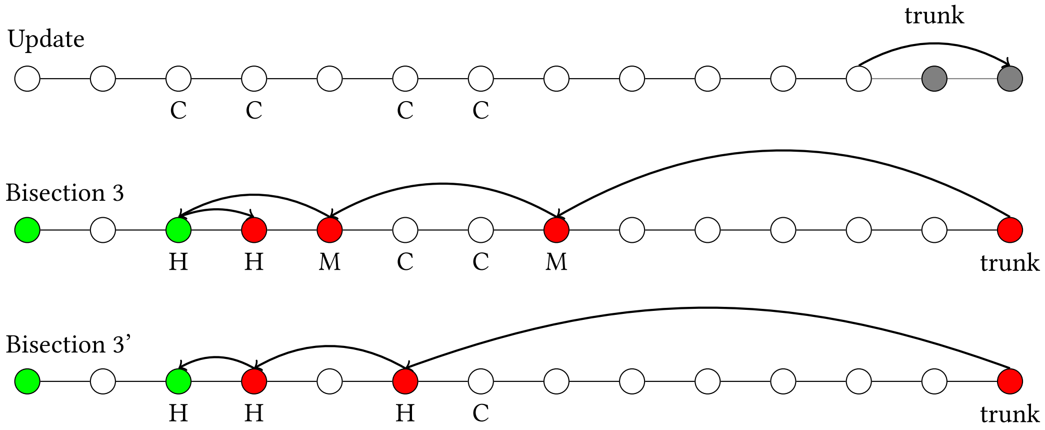


Fig. 3. Continuation of Figure 2. Due to the extension of the history in the first step, the bisection algorithms midpoints shift. *Bisection 3* bisects to the same commit as *Bisection 1* but has cache misses because of the shifted midpoints. Furthermore, the old midpoint is now very unlikely to be used in a bisection again due to its proximity to the new midpoint. The same bisection is run in *Bisection 3'* with the **cache-informed** algorithm. As the regression commit has already been bisected, we see a 100% hit rate. Bisecting a commit that has already been bisected is by far the most frequent bisection situation (subsection 4.3.)

[3] `git rev-list ^releases/gcc-11.2.0 master | wc -l` is 7184.
[4] `git rev-list ^llvmorg-4.0.0 main | wc -l` is 160328.

3.4 Reducer

The reducer is a wrapper around CReduce (subsection 2.5). It prepares the interesting candidate and the checker and then invokes CReduce. Often the reduced code can be simplified further manually. We dub this process **massaging**. Anecdotally, massaging candidate of bisection commits that are found often is more forgiving than massaging infrequently found bisection commits, where even swapping the order of unrelated statements can render the candidate uninteresting or change the bisection commit.

It is interesting to note that the bisection commit of an interesting candidate sometimes changes during the reduction. This is unwanted behavior but may indicate a closeness between these two regressions. Additionally, it is also possible for the set of attackers which can eliminate the marker of the candidate to change. A particularly noteworthy situation occurs when the set of interesting attackers changes but the bisection commit stays the same, hinting towards multiple regressions being introduced by the same commit. Analyzing candidates resulting from such situations remains to be studied.

3.5 Builder and Patcher

A lot of compilers need to be built when bisecting interesting candidates. Despite the efforts to reduce the number of compilers needed, DEAD has 5421 different compilers in its cache at the time of writing. To facilitate the building process, we implemented the builder which, given a compiler project and a commit-hash of the project, builds the compiler at that commit. Fortunately, the steps to build GCC and LLVM are stable, requiring only a one-time implementation for DEAD.

One problem with building older compiler versions lies in their dependencies. We encountered that a Linux header file older versions depend on was removed, function signatures changed or tests broke because they relied on faulty behavior in the dependency which was fixed in the meantime. We only required 7 different patches to build nearly all commits from LLVM `4.0.1` and GCC `7.5.0` up to their respective trunks.

Patcher This brings us to a different problem: given a commit and a set of patches, which patches are required to build this commit? A greedy building technique is too expensive as there are up to[5] $2^{|patches|}$ possible combinations to test. Applying all patches that can be applied and declaring defeat if it does not result in a successful build might apply unnecessarily many patches, potentially altering the behavior DEAD is testing. First trying to build the commit without any patches and then building with the patches is wasteful in several ways. The build failure may only reveal itself at the end of the building process, such as encountered with the broken unit test. Also, most commits need at least one patch, effectively making this approach equivalent to applying all applicable patches in the first place, but up to twice as computationally expensive.

The solution of the patcher is based on the insight that the commits that require a specific patch or patch combination forms a (mostly) continuous region in the git history. Given a commit c that fails to build without patches and a set of patches P that, when applied to the commit, result in a successful build, the patcher finds the continuous region around c that fails to build but builds when P is applied.

(1) Find the oldest commit oc that builds with but not without patches via bisection.
(2) For each branch that has oc as an ancestor, bisect the commit that last requires patches P to build.
(3) Construct and save the region which requires P to build from all bisected commits.

The initial P for c to successfully build has to be found manually. An example of running the patcher is shown in Figure 4. Each time a compiler is built, the builder first requests the list of all patches

[5] A working combination may be found earlier in the process.

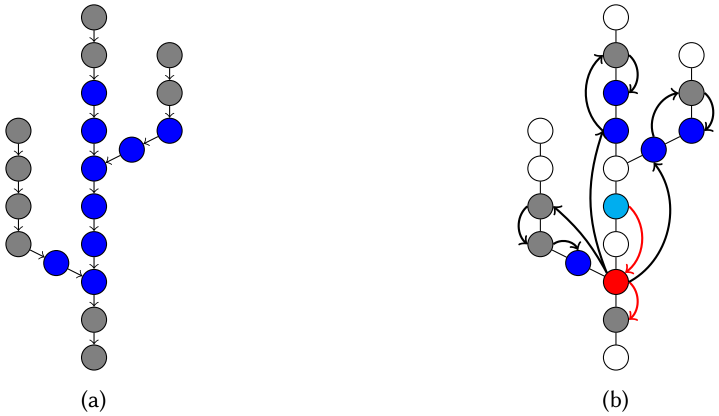


Fig. 4. Figure 4a shows a git history with the `parent` relation. Blue nodes indicate commits that require the patch, commits of gray nodes do not require the patch. How the patcher finds the blue region is shown in Figure 4b. Starting from the cyan-colored known broken but patchable node in the middle of the graph, the patcher first searches for the oldest commit that requires the patch, indicated by the red arrows. From this oldest patchable commit, the patcher runs a bisection for each branch that has the oldest patchable commit as its ancestor.

expected to be required from the patcher. In case the build fails despite the patches, the patcher saves this information in its database. Future attempts by the builder to build the faulty commit will be prevented by the patcher, increasing efficiency. All algorithms using the patcher need to be able to handle such a build failure. Throughout the usage of DEAD we encountered between 107 and 300 build failures proving the effectiveness of the patcher.

4 RESULTS & EVALUATION

In subsection 4.1 we will show examples of regressions found by DEAD.

In subsection 4.2 we are going to explore how dead code elimination evolves across compilers. To achieve this, we introduce a new "Advantage Analysis" which is based on DCE. We show that both LLVM and GCC are improving over time and, in general, show little regressions. Nevertheless, we find that LLVM has a non-trivial advantage over GCC in this analysis, although there is potential for both projects to learn from each other.

In subsection 4.3, we will analyze the 647 regressions found by DEAD and their relative frequency. We will see that, by the nature of unguided differential testing, it takes up to hundreds of attempts to find an interesting candidate which then bisects most likely to any of seven prevalent regression commits. Additionally, we will analyze which components of the compiler DEAD finds regressions in. Finally, we show a correspondence between the regressions found by DEAD and the underlying compiler structure.

4.1 Example Regressions

Listing 4 and Listing 5, and Listing 6 and Listing 7 are regressions found by DEAD for LLVM and GCC, respectively. Listing 4 shows a regression between `clang-trunk -Oz` and `clang-10.0.1 -Oz`. Note that in line 1 `a` is `static` and thus initialized to 0. On line 6, the first line of `main`, `a` is tested to be equal to 2, if so, `foo` is called. Naturally, this is not the case. Surprisingly, LLVM as a state-of-the-art compiler fails to eliminate this call.

Listing 6 shows a regression for `gcc-trunk -O3` against its latest release `11.2.0`. There are two opportunities to eliminate `foo`. First, the `for`-loop is never entered as `a = 0`. This was reported in Bug 99357. The second opportunity is to infer that `*c = 1` and thus `!(2 » (1/*c))` evaluates to 0. We note that Listing 4 and Listing 7 were *not* massaged, displaying the effectiveness of CReduce.

```
1  static char a;
2  short b, d;
3  static int c = 2;
4  void foo();
5  int main() {
6    if (a == 2) {foo();}
7    a = 9;
8    for (;; --a)
9      if (c) break;
10   d = b = 9;
11   if ((5 <= b | 1) % 12) {}
12   else {c = 0;};
13 }
```

Listing 4. `clang-trunk -Oz` could not eliminate `foo` while `clang-10.0.1 -Oz` could. We can clearly see that a is 0 at the beginning of `main`. (Issue 53317 on GitHub)

```
static int a[] = {1};
static int *b = &a[0];
static int **c = &b, **d = &b;
void foo(void);
static int *e() {
  for (;;) {
    if (a[0]) break;
    if (b) foo();
  }
  return *d;
}
int main() { *c = e(); }
```

Listing 5. `clang-trunk -O3` could not eliminate `foo` but `clang-13.0.0 -O3` could. `a[0]` is never written and is thus stays 1. (Issue 53322 on GitHub)

```c
void foo(void);
static int a, b = 1, *c = &b;
int main() {
  for (; a; a--) {
    int d = 2 >> (1 / *c);
    if (!d) foo();
  }
}
```

Listing 6. gcc-trunk -O3 could not eliminate foo but gcc-11.2.0 -O3 could. (Bug 104526)

```c
void foo();
int main() {
  unsigned a = -23;
  for (; a >= 2; ++a)
    if (a == 55)
      foo();
}
```

Listing 7. gcc-trunk -Os could not eliminate foo but gcc-11.2.0 -Os could. (Bug 105086)

4.2 Dead Code Elimination across Compilers

Compilers are constantly changing due to the continuous work on them. We want to analyze how these changes manifest with respect to dead code elimination. We aim to answer:

- Are compilers improving in their ability to eliminate code?
- Are regressions being fixed across versions?
- How do regressions evolve?
- What impact do different optimization levels have on DCE?
- How do GCC and LLVM compare in their ability to eliminate code?

We tracked markers across compiler versions and optimization levels in a set of generated files. The files were generated by running the file generation function of the generator (subsection 3.1) Note that we are *not* testing for interestingness during the generation to prevent a potential selection bias. Each file is then instrumented with *unique* markers i.e. no two different files share a marker and no two markers of one file are the same. We denote the set of all files with $\mathcal{F}$. Furthermore we denote the set of all compilers considered with C and their respective optimization levels with O. For the sake of simplicity, we ignore that LLVMs `Oz` has no correspondent in GCC in this notation.

The extraction of the markers is done as described in subsection 2.2: for each file, for each compiler version of interest and for each optimization level, the file is compiled with the specified compiler and optimization level, and the remaining (alive) markers are extracted from the produced assembly. We call this procedure $get\_alive(c, o, f)$ for $(c, o, f) \in C \times O \times \mathcal{F}$.

Some markers can not be removed as they, for example, lie in a part of the program which is guaranteed to be executed. Other markers might always be removed by any compiler that performs dead code elimination, for example, `if (false) marker();`. When analyzing the evolution of DCE, these markers will not provide any additional insight and thus should be removed. To capture this formally, we introduce:

Definition 4.1. **Trivial Marker**: A marker is considered to be **trivial** with respect to a set of compilers C and their optimization levels O iff it is either always or never eliminated by any $(c, o) \in C \times O$. The set of all trivial markers is denoted with TM.

Definition 4.2. **Non-Trivial Marker**: A marker is non-trivial iff it is not trivial. Concretely this means that there is at least one compiler-optimization level-combination that eliminates the marker and one which does not. We denote the set of all non-trivial markers with NTM.

One can compute NTM via the following relation:

$$NTM = \bigcup_{f \in F} \left(\underbrace{\bigcup_{(c,o) \in C \times O} get\_alive(c,o,f)}_{\text{Ever alive}} - \underbrace{\bigcap_{(c,o) \in C \times O} get\_alive(c,o,f)}_{\text{Never eliminated}} \right)$$

This relies on the fact that "Ever alive" will not contain markers that are always eliminated and thus never alive. In our experiments, the size of NTM was 66615. To be able to compare compilers with each other we need for each compiler c the subset of NTM which can be eliminated by c. We will refer to these sets by NTM_c. Obtaining NTM_c by conditioning the definition of NTM on $C = \{c\}$ does *not* yield the desired set as "Ever alive" will not include a marker m of a file f which is never alive *when using* c but $\exists (c', o) \in C \backslash \{c\} \times O : \; m \in get\_alive(c', o, f)$. Hence we define NTM_c as follows:

$$NTM_c = NTM - \underbrace{\bigcup_{f \in \mathcal{F}} \bigcap_{o \in O} get\_alive(c,o,f)}_{\text{Never eliminated when using } c}$$

For fixed compiler c and optimization level o:

$$NTM_{c,o} = NTM - \bigcup_{f \in \mathcal{F}} get\_alive(c,o,f)$$

Naturally, the question arises of how one can compare these metrics across compilers and optimization settings. We define the *advantage* one compiler-optimization level pair (Setting) has over another one as follows.

Definition 4.3. **Advantage**: For a reference set R and two sets A and B, the advantage of A *over* B with respect to R is

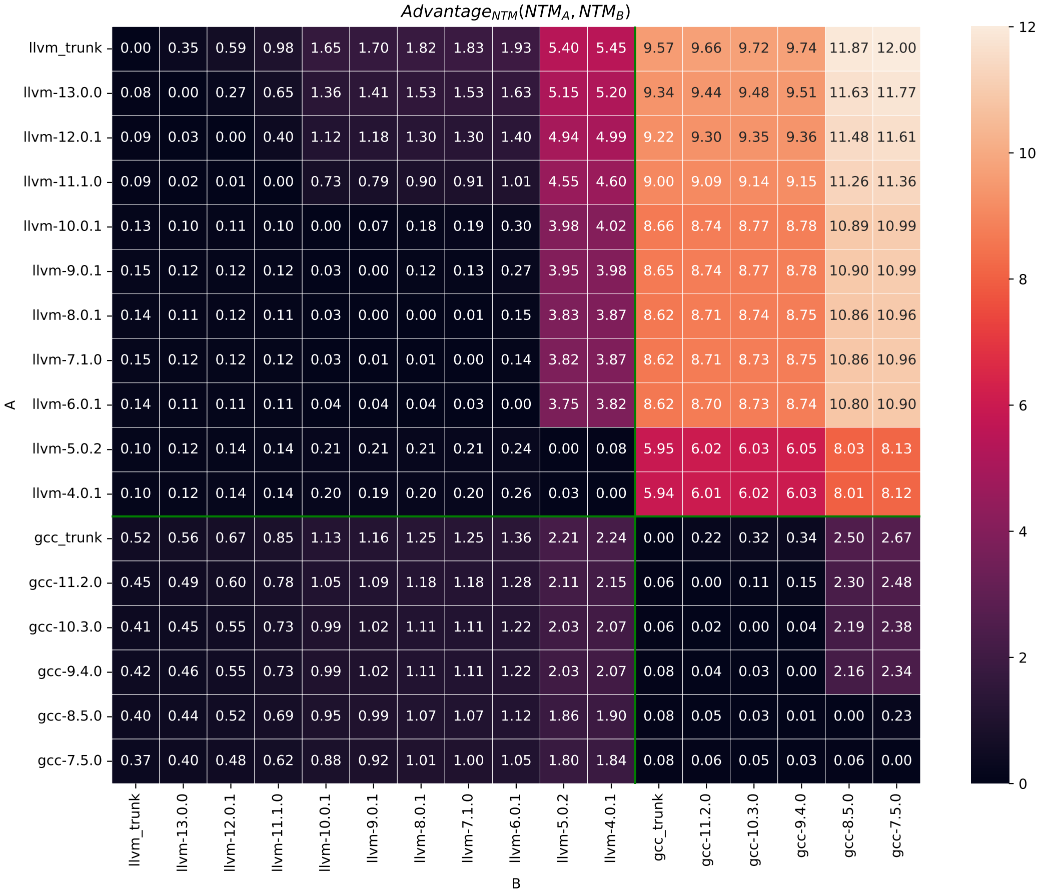
$$Advantage_R(A, B) = \frac{|A - B|}{|R|} \cdot 100$$

Note that we do not impose any restrictions on the relationship of A, B and R as R is just used as a normalizing constant. If, however, $A, B \subseteq R$ the advantage shows a direct percentage of R e.g. for two compilers $c, c' \in C$, the advantage of c over c' w.r.t. NTM $Advantage_{NTM}(NTM_c, NTM_{c'})$ is the percentage of NTM that c found but c' did not.

Compiler Advantage Comparison In the first experiment depicted in Figure 5 we compute the advantage of all compiler versions against each other with NTM as the reference set. As $\forall c \in C \; NTM_c \subseteq NTM$, the resulting values can be interpreted as direct percentages of NTM. The axes are lexicographically sorted by first the compiler project and then the compiler version. We will refer to such plots as "Advantage plots".

The setup of the experiment naturally divides the advantage plot into 4 regions as indicated by the green lines. In the bottom left and top right we find the advantages of GCC over LLVM and LLVM over GCC respectively. In the top left and bottom right, the compilers compare against themselves.

LLVM is getting better over time: Reading the upper right triangle of the upper left region in Figure 5 from left to right i.e. comparing the same version against increasingly older versions, shows an increase in advantage. For LLVM at trunk, we see an advantage from 0.35 when comparing against version `13.0.0` up to an advantage of 5.45 against `4.0.1`. In the lower-left triangle of the upper-left region we compare older compiler versions against newer ones, representing regressions



A \ B	llvm_trunk	llvm-13.0.0	llvm-12.0.1	llvm-11.1.0	llvm-10.0.1	llvm-9.0.1	llvm-8.0.1	llvm-7.1.0	llvm-6.0.1	llvm-5.0.2	llvm-4.0.1	gcc_trunk	gcc-11.2.0	gcc-10.3.0	gcc-9.4.0	gcc-8.5.0	gcc-7.5.0
llvm_trunk	0.00	0.35	0.59	0.98	1.65	1.70	1.82	1.83	1.93	5.40	5.45	9.57	9.66	9.72	9.74	11.87	12.00
llvm-13.0.0	0.08	0.00	0.27	0.65	1.36	1.41	1.53	1.53	1.63	5.15	5.20	9.34	9.44	9.48	9.51	11.63	11.77
llvm-12.0.1	0.09	0.03	0.00	0.40	1.12	1.18	1.30	1.30	1.40	4.94	4.99	9.22	9.30	9.35	9.36	11.48	11.61
llvm-11.1.0	0.09	0.02	0.01	0.00	0.73	0.79	0.90	0.91	1.01	4.55	4.60	9.00	9.09	9.14	9.15	11.26	11.36
llvm-10.0.1	0.13	0.10	0.11	0.10	0.00	0.07	0.18	0.19	0.30	3.98	4.02	8.66	8.74	8.77	8.78	10.89	10.99
llvm-9.0.1	0.15	0.12	0.12	0.12	0.03	0.00	0.12	0.13	0.27	3.95	3.98	8.65	8.74	8.77	8.78	10.90	10.99
llvm-8.0.1	0.14	0.11	0.12	0.11	0.03	0.00	0.00	0.01	0.15	3.83	3.87	8.62	8.71	8.74	8.75	10.86	10.96
llvm-7.1.0	0.15	0.12	0.12	0.12	0.03	0.01	0.01	0.00	0.14	3.82	3.87	8.62	8.71	8.73	8.75	10.86	10.96
llvm-6.0.1	0.14	0.11	0.11	0.11	0.04	0.04	0.04	0.03	0.00	3.75	3.82	8.62	8.70	8.73	8.74	10.80	10.90
llvm-5.0.2	0.10	0.12	0.14	0.14	0.21	0.21	0.21	0.21	0.24	0.00	0.08	5.95	6.02	6.03	6.05	8.03	8.13
llvm-4.0.1	0.10	0.12	0.14	0.14	0.20	0.19	0.20	0.20	0.26	0.03	0.00	5.94	6.01	6.02	6.03	8.01	8.12
gcc_trunk	0.52	0.56	0.67	0.85	1.13	1.16	1.25	1.25	1.36	2.21	2.24	0.00	0.22	0.32	0.34	2.50	2.67
gcc-11.2.0	0.45	0.49	0.60	0.78	1.05	1.09	1.18	1.18	1.28	2.11	2.15	0.06	0.00	0.11	0.15	2.30	2.48
gcc-10.3.0	0.41	0.45	0.55	0.73	0.99	1.02	1.11	1.11	1.22	2.03	2.07	0.06	0.02	0.00	0.04	2.19	2.38
gcc-9.4.0	0.42	0.46	0.55	0.73	0.99	1.02	1.11	1.11	1.22	2.03	2.07	0.08	0.04	0.03	0.00	2.16	2.34
gcc-8.5.0	0.40	0.44	0.52	0.69	0.95	0.99	1.07	1.07	1.12	1.86	1.90	0.08	0.05	0.03	0.01	0.00	0.23
gcc-7.5.0	0.37	0.40	0.48	0.62	0.88	0.92	1.01	1.00	1.05	1.80	1.84	0.08	0.06	0.05	0.03	0.06	0.00

Fig. 5. Each cell shows the advantage of A (y-axis) over B (x-axis). The different compiler projects divide the grid into four regions, indicated by the green lines. LLVM finds more total markers than GCC in all cases. We see improvements in both compiler projects. This is particularly noticable when transitioning from GCC `8.5.0` to `9.4.0` and LLVM `5.0.2` to `6.0.1`. LLVM is slowly extending its advantage over GCC while GCC's advantage over LLVM is slowly decreasing. Both projects only show small amounts of regressions. Advantage plots specific to optimization levels can be found in Appendix A.

(regression values). We can see that these values in general are smaller than the values on the upper-right triangle, indicating that LLVM does find new markers over time while having fewer regressions than new markers eliminated. Additionally, the regression values are stable across versions, indicating that few new regressions are being introduced. For versions `4.0.1` and `5.0.2` we even find a reduction in regression values since version 11, showing that regressions, knowingly or unknowingly, are being fixed. We see one notable jump in advantage from version `5.0.2` to `6.0.1` of 3.75.

GCC is getting better over time: This can be seen in the upper right triangle of the lower right region. GCC at trunk has an advantage of 0.22 against version `11.2.0` which increases to 2.67 against `7.5.0`. The regression values of GCC are particularly low at a range of $0.01 - 0.08$, indicating very little regression.

LLVM has a noticeable advantage over GCC: LLVM has an advantage of at least 5.9 in any pairing i.e. finds at least 5.9% of all markers, that GCC does not find. Perhaps unsurprisingly, the highest advantage of 12 is found when comparing the current trunk of LLVM against the oldest version of GCC in the analysis, `7.5.0`. When dropping the two oldest versions of each compiler, the advantage range of LLVM over GCC shrinks to $8.6 \leq x \leq 9.7$. Comparing both trunks gives 9.57, a value close to the top of the advantage range of LLVM over GCC. We can also see that in recent versions LLVM is increasing its advantage value over GCC. LLVM `6.0.1` has an advantage over GCC `9.4.0` of 8.74, LLVM `9.0.1` against GCC `10.3.0` shows next to no increase with 8.77, LLVM `12.0.1` against GCC `11.2.0` increases the advantage to 9.3 and LLVM trunk against GCC trunk further increases this to 9.57. Note that the initial releases of the oldest versions of both compiler projects are close to each other with LLVM `4.0.0` being released on 13.03.2017 and GCC `7.1` with a release date of 02.05.2017. Thus the development time for both projects is roughly equivalent. However, LLVM received 4.25 times as many commits as GCC during this period.
GCC's small advantage over LLVM is decreasing: On the other side, the advantage range of GCC over LLVM is $0.37 \leq x \leq 2.24$ with all compilers and $0.40 \leq x \leq 1.36$ when dropping the two oldest versions of each. Again comparing the trunks yields 0.52, a value close to the bottom of the range. Also, the advantage has been decreasing over the last versions albeit very slowly ($1.12 \rightarrow 0.52$) compared to the total advantage of LLVM over GCC.
Both compiler projects show little regression: In the regions where the compiler projects compare against themselves, we see a constant improvement in both projects as the values decrease when reading from right to left in the respective upper right triangle. Similarly, we can estimate the development of regressions by reading the lower-left triangle from top to bottom. Again, both projects show positive results by having low, near-constant values across the board.

Both projects also have one significant jump between versions. LLVM improved by around 3.5 from version `5.0.2` to `6.0.1` and GCC by 2.2 from version `8.5.0` to `9.4.0`. Another observation to note here is that LLVM enabled a new passmanager between versions 12 and 13. An event that does not stick out in this plot, indicating it was well tested.

Figure 5 raises another question: If we can interpret the values as percentages of NTM and the highest value is 12, where are the other 88% of markers? One possible explanation could be that the compilers drastically change the markers they find across versions. But we have seen that both projects have a small changing set of markers because both their rate of improvement and the amount of regressions are small (compared to 88%). The problem is that in Figure 5, each compiler competes with NTM_c which can hide a lot of markers in the intersection which will show up in NTM, as there is at least one compiler that could not eliminate this.

Plotting the elimination percentage of each $(c, o) \in C \times O$ (Figure 6) shows particularly low percentages for settings with `O1`. Also `GCC 8.5.0 -Os` with 78.6% eliminations and `GCC 7.5.0 -Os` with 36.7%(!) are part of the same region. On an interesting side note, the elimination percentage between `LLVM 12.0.1 -O1` and `LLVM 13.0.0 -O1` increased by 37.2% from a particularly low 59.7% to 96.9%, which beats the best GCC setting by 6%. We believe the swap to the new passmanager might be the cause for this.

Compiler advantage comparison on unique marker groups The observations above suggest that there are markers in NTM which are eliminated or missed because of the presence or absence of the same type of analysis or feature, leading to whole groups of markers being eliminated or missed at once. To focus on the presence or absence of these features, we want to restrict our analysis to one marker of each group. To distinguish markers we define:

Definition 4.4. **Signature**: The signature of a marker m given a total ordering Or of $C \times O$ is the bit-vector we get from testing if $m \in get\_alive(c, o, f)$ for each $(c, o) \in C \times O$ in the order

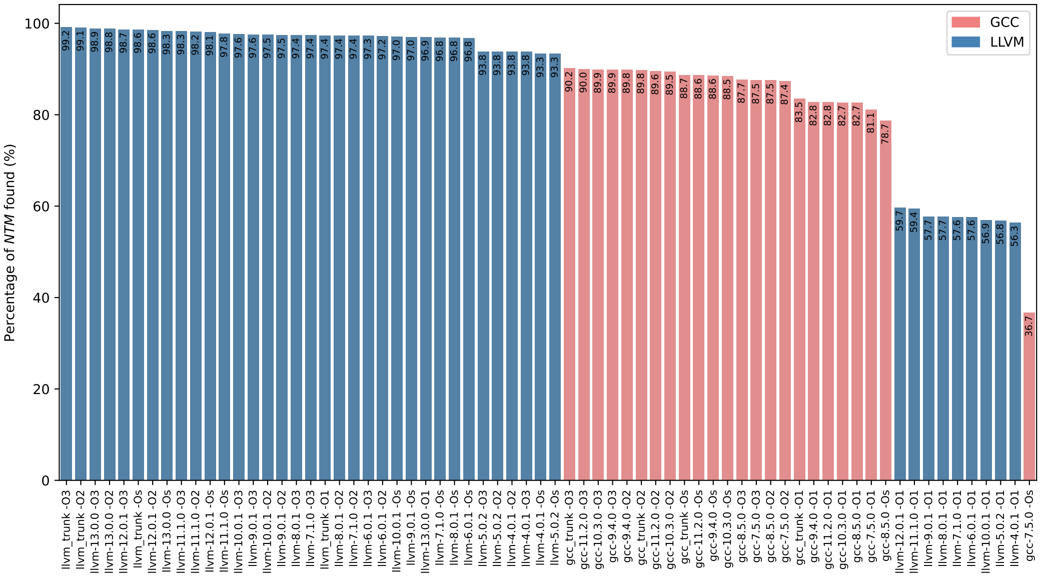


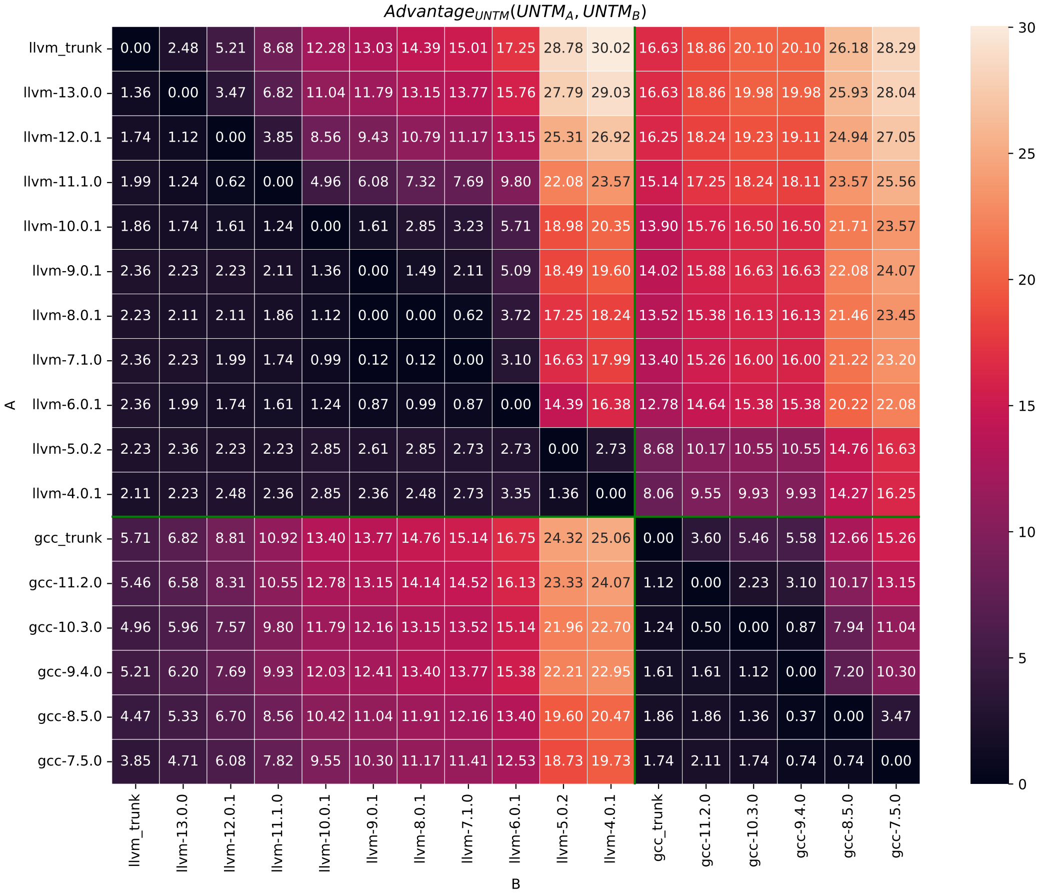
Fig. 6. LLVM outperforms GCC on *NTM* with nearly all optimization levels. LLVM O1 made an improvement of 37.2% from version `12.0.1` (59.7%) to version `13.0.0` (96.9%). Similarly, GCC's Os made a 42% improvement from version `7.5.0` (36.7%) to `8.5.0` (78.7%). It is interesting to note that the worst performance is from Os, as both projects state that Os is based on O2 which outperforms Os. Furthermore, GCC's O3 of one version performs better than some of the next versions at O2 while with LLVM the compiler version is more important than the optimization level. A version of the plot containing all optimization levels can be found in Figure 15.

defined by Or. As the set of markers for a file is independent between files, there is only one f which includes m. We will refer to this by $sig_{C,O}(m)$. We also define this operation for sets. For a set of markers M, the set of all signatures found in M is $sig_{C,O}(M)$. We will omit C and O when they are clear from the context.

The idea is that markers with a different signature must be missed or eliminated by a different set of features, otherwise, the marker would have the same signature. Note that markers with the same signature must not be caused by the same set of features. Partitioning NTM based on different signatures shows that there are 806 unique groups. The largest group accounts for 41% of NTM and with `GCC 7.5.0 -Os` as the only setting failing to eliminate them and the second largest group including 26% of NTM is caused by LLVM versions in O1 before version 13. For further analysis we define

Definition 4.5. **Unique Non-Trivial Marker**: A unique non-trivial marker m in a set of non-trivial markers M derived from compilers C, optimization levels O and files $\mathcal{F}$, is a non-trivial marker such that $\forall m' \in M \backslash \{m\} : \ sig(m) \neq sig(m')$, i.e., has a unique signature. The set $UNTM$ is then a set of markers with $UNTM \subseteq M$, $sig(UNTM) = sig(M)$ and where each marker in $UNTM$ is unique. Note that by construction $UNTM$ might not be unique but rather gives a representation of the unique partition of M based on the equivalence relation induced by signature equality.

Figure 7 shows the advantage plot based on $UNTM$ values. Most of the findings of Figure 5 can still be observed here however some changes emerged:



	llvm_trunk	llvm-13.0.0	llvm-12.0.1	llvm-11.1.0	llvm-10.0.1	llvm-9.0.1	llvm-8.0.1	llvm-7.1.0	llvm-6.0.1	llvm-5.0.2	llvm-4.0.1	gcc_trunk	gcc-11.2.0	gcc-10.3.0	gcc-9.4.0	gcc-8.5.0	gcc-7.5.0
llvm_trunk	0.00	2.48	5.21	8.68	12.28	13.03	14.39	15.01	17.25	28.78	30.02	16.63	18.86	20.10	20.10	26.18	28.29
llvm-13.0.0	1.36	0.00	3.47	6.82	11.04	11.79	13.15	13.77	15.76	27.79	29.03	16.63	18.86	19.98	19.98	25.93	28.04
llvm-12.0.1	1.74	1.12	0.00	3.85	8.56	9.43	10.79	11.17	13.15	25.31	26.92	16.25	18.24	19.23	19.11	24.94	27.05
llvm-11.1.0	1.99	1.24	0.62	0.00	4.96	6.08	7.32	7.69	9.80	22.08	23.57	15.14	17.25	18.24	18.11	23.57	25.56
llvm-10.0.1	1.86	1.74	1.61	1.24	0.00	1.61	2.85	3.23	5.71	18.98	20.35	13.90	15.76	16.50	16.50	21.71	23.57
llvm-9.0.1	2.36	2.23	2.23	2.11	1.36	0.00	1.49	2.11	5.09	18.49	19.60	14.02	15.88	16.63	16.63	22.08	24.07
llvm-8.0.1	2.23	2.11	2.11	1.86	1.12	0.00	0.00	0.62	3.72	17.25	18.24	13.52	15.38	16.13	16.13	21.46	23.45
llvm-7.1.0	2.36	2.23	1.99	1.74	0.99	0.12	0.12	0.00	3.10	16.63	17.99	13.40	15.26	16.00	16.00	21.22	23.20
llvm-6.0.1	2.36	1.99	1.74	1.61	1.24	0.87	0.99	0.87	0.00	14.39	16.38	12.78	14.64	15.38	15.38	20.22	22.08
llvm-5.0.2	2.23	2.36	2.23	2.23	2.85	2.61	2.85	2.73	2.73	0.00	2.73	8.68	10.17	10.55	10.55	14.76	16.63
llvm-4.0.1	2.11	2.23	2.48	2.36	2.85	2.36	2.48	2.73	3.35	1.36	0.00	8.06	9.55	9.93	9.93	14.27	16.25
gcc_trunk	5.71	6.82	8.81	10.92	13.40	13.77	14.76	15.14	16.75	24.32	25.06	0.00	3.60	5.46	5.58	12.66	15.26
gcc-11.2.0	5.46	6.58	8.31	10.55	12.78	13.15	14.14	14.52	16.13	23.33	24.07	1.12	0.00	2.23	3.10	10.17	13.15
gcc-10.3.0	4.96	5.96	7.57	9.80	11.79	12.16	13.15	13.52	15.14	21.96	22.70	1.24	0.50	0.00	0.87	7.94	11.04
gcc-9.4.0	5.21	6.20	7.69	9.93	12.03	12.41	13.40	13.77	15.38	22.21	22.95	1.61	1.61	1.12	0.00	7.20	10.30
gcc-8.5.0	4.47	5.33	6.70	8.56	10.42	11.04	11.91	12.16	13.40	19.60	20.47	1.86	1.86	1.36	0.37	0.00	3.47
gcc-7.5.0	3.85	4.71	6.08	7.82	9.55	10.30	11.17	11.41	12.53	18.73	19.73	1.74	2.11	1.74	0.74	0.74	0.00

Fig. 7. Advantage plot for compilers with *UNTM* as reference set. The findings of Figure 5 are mostly present here. However we see a significant increase of the advantage of GCC trunk over LLVM trunk to 5.71.

- **Significantly increased advantage of GCC over LLVM.** GCC trunks advantage over LLVM trunk grew from 0.52 to 5.71.
- **More significant releases for LLVM.** While the previously identified advantage from version `5.0.2` to `6.0.1` is still the most significant at 14.39 version over version. We see that the last three releases are showing an advantage of at least 3.47 over their predecessor.
- **GCC `10.3.0` performs worse than GCC `9.4.0`.** `10.3.0` has an advantage of 0.87 over `9.4.0`, while `9.4.0` has an advantage of 1.12 resulting in the only worse total advantage of a successor over its predecessor. We can see this manifesting itself in 8, where `9.4.0` is ranked higher than `10.3.0`. Note that we did not see this in Figure 5.

The updated and reduced percentage plot in Figure 8 shows that GCC is more competitive than in Figure 6 with GCC trunk being on par with LLVM `9.0.1` at ≈ 80.5% total elimination. We can also see the aforementioned significant releases of LLVM, pulling ahead of the competition since version `11.1.0`. Finally, the best contestant reaches 91.2%, leaving 8.8% of *UNTM* to be fixed/found; a promising sign for the usage of DEAD.

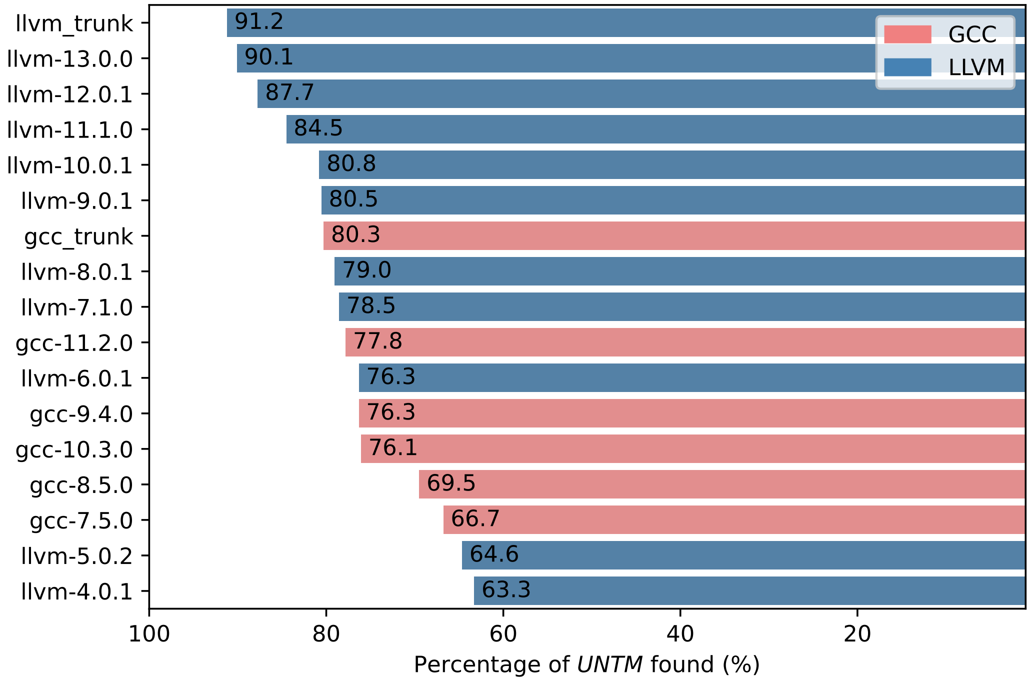


Fig. 8. Updated percentage plot for *UNTM*. Compilers are generally improving, with only GCC `10.3.0` having a lower score than its predecessor. LLVM is outperforming GCC by 10% at trunk, a lead which has been extended over the last releases. A version of the plot containing all optimization levels and compilers can be found in Figure 15.

4.3 Regression Analysis

To assess the strengths and weaknesses of DEAD, we want to analyze its behavior and the regressions found by it. We aim to answer:

- How long does it take to find a regression?
- In which part of the compiler did DEAD find regressions?
- In which optimization levels did DEAD find regressions?

From the high-level flow chart in Figure 1 we know that it can take several attempts to find an interesting candidate. DEAD records how many attempts it took to find a given candidate. It shows that it takes on average 109 attempts to find an interesting candidate and that GCC and LLVM do not require the same amount of attempts to find a candidate. In Figure 9, we can see the estimated cumulative density function (ECDF) for the attempts until an interesting candidate was found. 95% of interesting candidates were found after 197 attempts for GCC and after 753 attempts for LLVM. The maximum amounts of attempts are 861 and 3477 for GCC and LLVM respectively. Effectively, this means that it is easier to find an interesting candidate in GCC.

Regression Frequency Not every new interesting candidate found constitutes a new regression. In fact, only a small fraction of them are. DEAD distinguishes candidates by their bisected regression commit and saves, how often each regression is found. The pie chart in Figure 10 shows how often the 346 and 301 different regressions for LLVM and GCC, respectively, were found relative to each other. More than half of all interesting candidates for LLVM bisect to the top four most frequently found commits (Figure 10a). The commits and their changes are listed in Table 1.

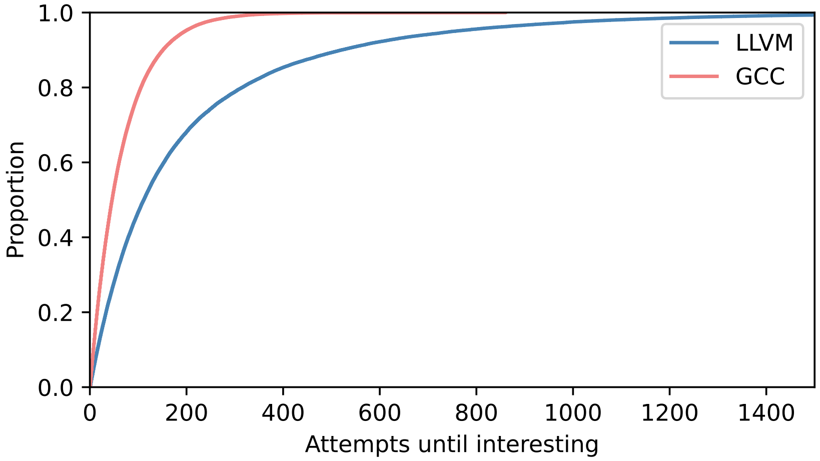
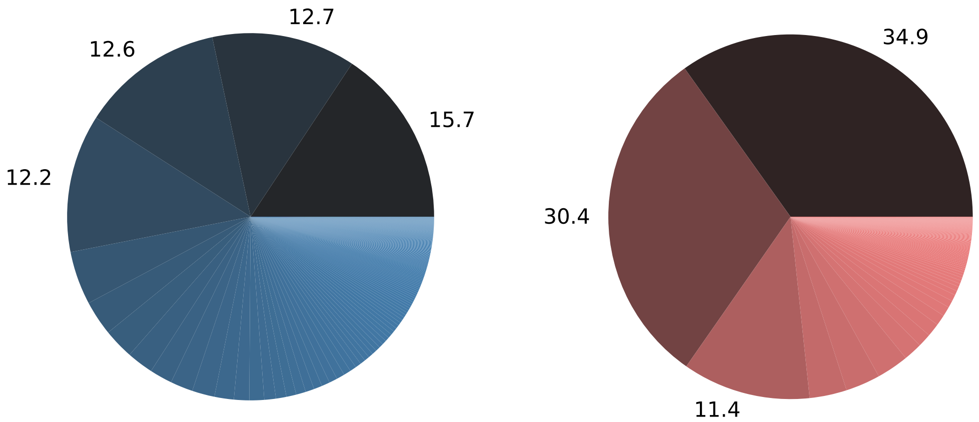


Fig. 9. Estimated cumulative density function (ECDF) for attempts needed by the generator to find an interesting candidate as shown the flow chart in Figure 1. Most interesting candidates are found after 300 attempts. We can also see that finding an interesting candidate takes longer on average for LLVM than GCC.



(a) LLVMs top 4 most frequently found regressions are `a6492e2271`,`669ddd1e9b`,`0ab0c1a201` and `51b809bf2f` and cover more than 50% of all interesting candidates found.

(b) GCC's top 3 most frequently found regressions in descending order are `0b92cf305d`, `d8edfadfc7` and `49d9c9d283`, together covering more than 75% of all interesting candidates found for GCC.

Fig. 10. Pie diagrams for LLVM and GCC showing how often different regression commits were found. We equate the importance of a regressions with how often it is found and thus how difficult it is to find. As the top most frequently found regressions of GCC cover more than 75% of all regressions found, we believe them to be a major factor of why it is easier to find a new interesting candidate for GCC compared to LLVM (Figure 9).

%	Commit	First Included	Changes
15.7	a6492e2271	11.0.0	Adds special case for alignment of constant pointers.
12.7	669ddd1e9b	13.0.0	Enables the new passmanager of LLVM.
12.6	0ab0c1a201	6.0.0	Changes sinking behavior with the intend to improve common subexpression elimination gains.
12.2	51b809bf2f	6.0.0	Changes the inlining heuristics for call-sites.

Table 1. Top four most frequently found regression commits for LLVM. Together, they cover more than 50% of all interesting candidates found for LLVM. Note that all these commits were first included in major versions that all perform considerably *better* than their predecessors (Figure 8).

We observe the following:

- All commits first appear in major versions which *improve* DCE according to subsection 4.2.
- Most changes made are expected to introduce regressions.

For GCC, the top three most frequently found cover more than 75%(!) of all interesting candidates found (Figure 10b). We believe this to be a major reason for the discrepancy between GCC and LLVM in the difficulty of finding interesting candidates in Figure 9. The changes for the top three regression commits are summarized in Table 2. We can observe the following for the top regressions of GCC:

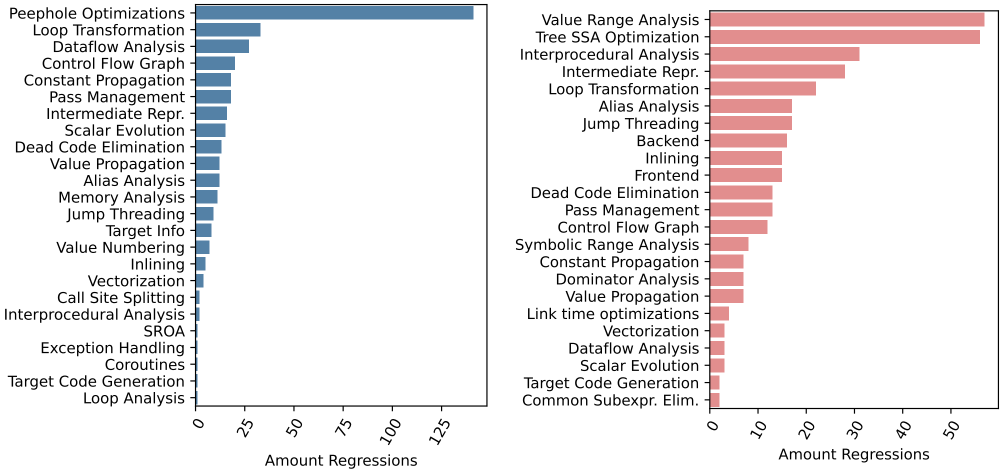
%	Commit	First Included	Changes
34.9	0b92cf305d	10.0.0	Early inlining decisions in 02
30.4	d8edfadfc7	master	Disallows loop rotation and loop header crossing in jump threaders.
11.4	49d9c9d283	10.0.0	Inlining analysis

Table 2. Top three most frequently found regression commits for GCC. They cover more than 75% of all interesting candidates found for GCC. Two of the three commits were first included with `10.0.0`, the only version that performs *worse* than its predecessor (Figure 8).

- Changes to inlining behavior gives rise to 46% of all interesting candidates found.
- Both commits changing inlining behavior were first released in GCC `10`, which is the only major release that performs slightly *worse* than its predecessor in advantage analysis of $UNTM$ in Figure 8.
- The author of commit `d8edfadfc7` feels "a bit queasy about such a fundamental change wrt threading".

Across compilers, inlining decisions seem to have a significant effect on DCE, appearing four times in the most frequently found regressions for LLVM and GCC. Furthermore, as the top commits include larger changes to the compiler such as the enabling of the new passmanager for LLVM or changing the way jump threading is performed for GCC, we conclude that DEAD is an effective tool to test such changes.

Compiler Components Affected by Regressions To further investigate what kind of bugs DEAD finds, we analyzed and categorized all files that were changed in all regressions found. In Figure 11 we show how often a category appeared across the regressions, separate for each compiler project. Each regression can be part of multiple categories. As GCC and LLVM have



(a) Category distribution for LLVM

(b) Category distribution for GCC

Fig. 11. These plots show in which components of the compilers DEAD found regressions. While the regressions in LLVM clearly come from Peephole Optimizations (141 times found), the regressions in GCC cover a wider range of topics. Value Range Analysis and Tree SSA Optimization with 57 and 56 related regressions found are the most common categories for GCC. A recent overhaul of GCC's Jump Threading as found in Table 2 is also reflected in this plot.

different structures, some categories are compiler-specific and thus do not have an equivalent in the other project.

Figure 11a shows that 141 regressions found by DEAD performed changes related to peephole optimizations. This is approx. 4 times as frequent as the second most common category "Loop Transformations" with 33 related regressions. We can observe a long tail of similarly frequent categories such as Data and Control Flow Analysis, DCE, Constant and Value Propagation, and Alias Analysis. Perhaps surprisingly, the number of regressions related to Inlining is only 5, given the impact changes to inlining can have, as we have seen in the Regression Frequency paragraph. GCC's categories shown in Figure 11b are more evenly distributed. 57 and 56 regressions were found related to Value Range Analysis and Tree SSA Optimizations, respectively. Again, Inlining, with 15 related regressions, is not as prevalent as Table 2 might lead one to believe. Indicated by the commit message in `d8edfadfc7` from Table 2, GCC is currently changing how threaders work. With such an undertaking, regressions are to be expected. DEAD finds regressions related to threading, indicated by categories "Jump Threading" and "Loop Transformations", highlighting the usefulness of DEAD for testing major changes.

Optimization Level Frequency To see what optimization levels are affected by regressions found by DEAD, we checked for each optimization level (OL) by how many of the 647 regressions they are affected. Concretely, we tested for each OL and regression if the marker is alive in trunk and if there is an older version of the compiler that eliminates the marker on the *same* OL. Note that one regression may be present for multiple OL. We plotted the Venn diagram of the results for GCC and LLVM in Figure 12 and Figure 13, respectively. In both figures, we can find combinations of OL that contain no or very few regressions. e.g. there is only one regression that is present in only 01 and 02 for GCC. We can observe the following:

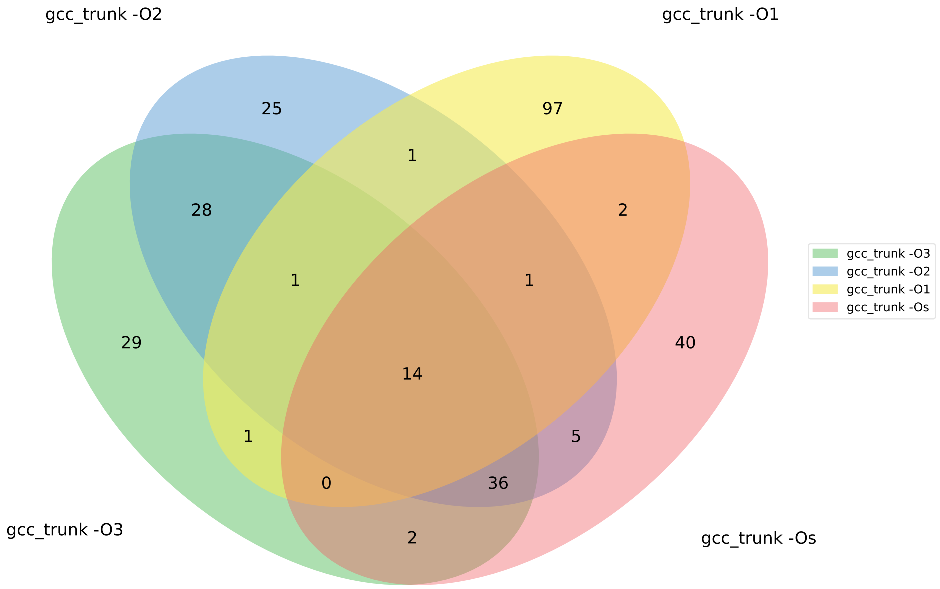


Fig. 12. Venn diagram of which optimization levels are affected by the regressions found for GCC. The OL combination with the most regressions at 97 is 01, followed by 0s with 40. In general, regressions for GCC seem to be more specific OL rather than affect all OL ,i.e., more regressions are found on the outer rim of ring of the diagram rather than in the center.

(1) **01 has a low overlap with other OL except when the regression is present in all OL.** This can be observed for both LLVM and GCC. We count 6 such regressions for GCC and 9 for LLVM.
(2) **The most shared regressions can be found in all OL excluding 01.** We find 36 such regressions for GCC and 89 for LLVM. This effect is more pronounced in LLVM, not only in total value but also when compared to all other values. This directly leads to the next observation.
(3) **Compared to LLVM, GCC shows more OL-specific regressions.** The ratio of regressions that are only found by one OL and regressions found by more than one OL is 0.57 for LLVM and 2.07 for GCC (1.03 without 01). We interpret this as greater independence between the different OL in GCC.
(4) **Many regressions found for GCC are only for 01.** With 97 regressions, GCC's 01 has more than double the amount of regressions than any other combination of OL for GCC.
(5) **LLVM has few regressions present in only 02 and only 0s.** They have 8 and 9 regressions respectively while other OL have at least 19.

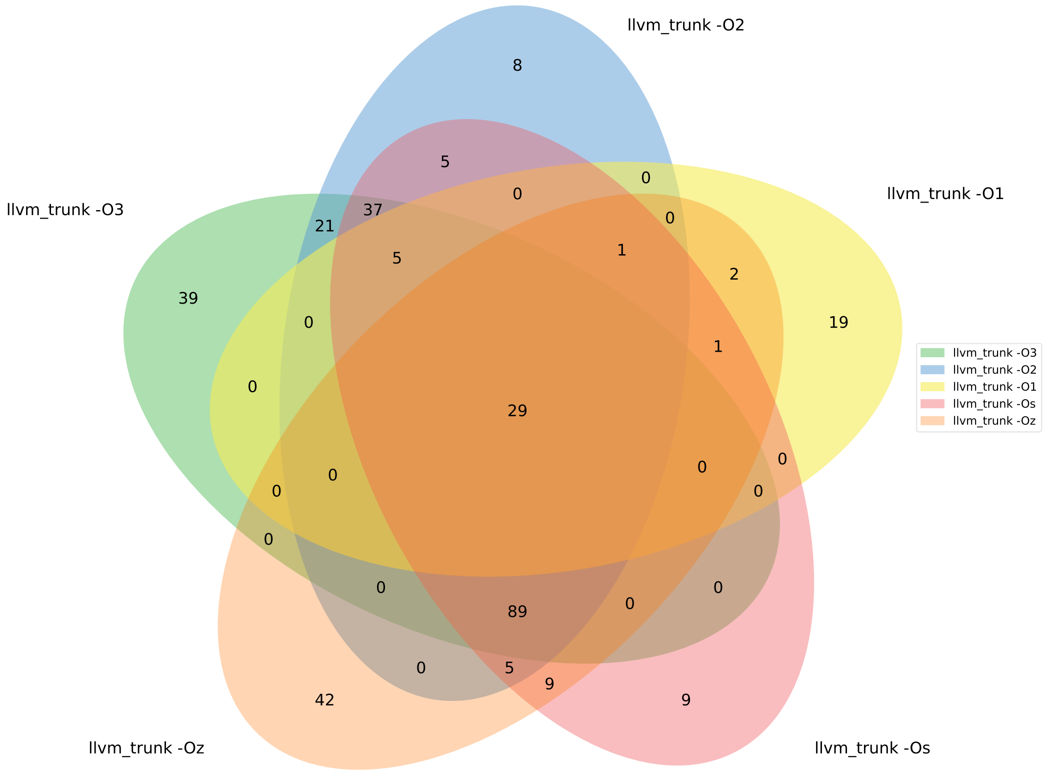


Fig. 13. Venn diagram of which optimization levels are affected by the regressions found for LLVM. The OL combination with the most regressions is O3, O2, Os with 89 regressions, followd by Oz (42), O3 (39) and O3, O2, Os with 37 regressions. Regressions for LLVM are more often shared between at least two OL compared to GCC.

The architecture of the optimization passes is reflected in the Venn diagrams. The man-page of clang describes the relationship between the different OL as follows:

- O1 is somewhere between O0 and O2.
- O2 corresponds to a moderate level of optimization which enables most optimizations.
- O3 is like O2 but enables additional, more expensive optimizations.
- Os is like O2, with extra optimizations for code size.
- Oz is like Os but reduces code size further.

From these descriptions, we can derive an estimate of how many regressions should be found for the OL combinations. We expect to find:

- some regressions which all OL miss as all OL are defined relative to O2
- more regressions found by all OL except O1 because most/more complicated optimizations are performed at and above O2.
- few regressions in O2 and Os alone as they are the basis for O3 and Oz respectively.
- few regressions for O1 only as it is described as a subset of O2.

A prediction about the relative amount of regressions for O3 and Oz can not be made as the scope of OL-specific optimizations is not given. Any other combination of OL is not expected to have an overlap. A visualization of the expected relations is shown in Figure 14.

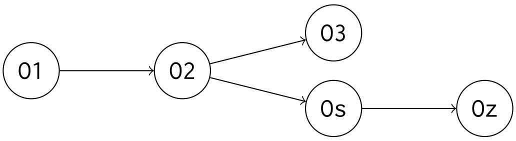


Fig. 14. Visualization of the expected subset relation for the optimization levels w.r.t. the found regressions, derived from the description in clangs man-page.

One may recognize some of the earlier observations in the list of expectations. Namely that LLVMs O2 and Os have few regressions specific to them (item 5), some regressions are found in all OL (item 1) ,and more regressions that affect all OL except O1 (item 2). We do *not* find that O1 has few specific regressions. Quite the opposite is the case for GCC (item 4) and also LLVM with 19 specific regressions for O1. We believe that as O2 performs more optimizations and is considered a safe choice to compile system-critical programs such as the Linux kernel, the use-cases for O1 in real software are few. Therefore, regressions in O1 are met with less scrutiny and are of less or no interest. O1 should be excluded when running DEAD for other purposes than curiosity or research. Furthermore, there is a considerable amount of regressions for O3 and O2 only in both GCC (21) and LLVM (28). GCC's man-page provides an explanation for its regressions: GCC's Os, although based on O2, disables some of the optimizations of O2 in favor of binary size. We assume LLVM to perform similar trade-offs although not explicitly mentioned.
For LLVM we also find a collection of 37 regressions specific to O3, O2, and Os, only missing Oz to belong to the largest patch of regressions. Similar to the regressions unique to O3 and O2, Oz seems to disable further optimizations compared to Os in favor of size.

In general, the regressions found by DEAD seem to reflect the underlying structure of the compilers. We conclude that DEAD could be used to study opaque compilers e.g. where only the binaries are available, highlighting a further use-case for the tool.

5 CONCLUSION

We successfully extended the work of [Theodoridis et al. 2022] and built DEAD.
We developed the *in-cache bisection* and saw its effectiveness when bisecting the many interesting candidates found by the generator. However, the bisection required many compilers to be built which were not readily buildable. To counter this issue, we developed the patcher which cheaply establishes the region where patches need to be applied for a successful build.

Furthermore, we analyzed the 647 regressions found by DEAD and identified, perhaps unsurprisingly, that the components affected by the regressions were most likely related to Peephole Optimizations in LLVM and Value Range Analysis and Tree SSA Optimizations in GCC. We also found that most interesting candidates bisected to one of seven commits, three of which are related to inlining. Additionally, we discovered that the regressions reflect the underlying structure of compilers, allowing the exploration of opaque,i.e., closed sourced compilers with DEAD.

Lastly, we proposed the metric of "Advantage", resulting in a novel analysis to compare different compilers and optimization levels. We used it to study the evolution of LLVM and GCC since 2017 and saw that both are improving their ability to eliminate dead code over time. Comparing LLVM and GCC showed that LLVM has a non-trivial advantage over GCC which has been slightly increasing over the last versions.

In conclusion, we showed that DEAD is an effective and versatile tool for finding missed opportunities, enabling promising future research.

5.1 Future Work

There are several ways to improve DEAD as a tool. Currently, DEAD does not verify that the transformation that previously allowed for the elimination of a marker, is correct. This could lead to false-positive reports, which can have devastating effects on the trust of compiler developers in DEAD. At the moment, the safety procedure is to manually check the commit message of the regression commit before reporting. Such verification could be implemented with the help of Alive2 of [Lopes et al. 2021]. Furthermore, DEAD does not leverage compiler-specific structures such as the passmanager and `llvm-reduce` of LLVM to generate more specific bug reports.
CSmith is the only program generator used to generate new candidates in DEAD. Different generators that target different parts of the compiler (e.g. interprocedural analysis) could be leveraged to find more specific bugs. Yarpgen by [Livinskii et al. 2020] was briefly tested, however, the amount of loops made testing the candidates slow. When different category-specific generators are in place, the testing capabilities of DEAD would also be more efficient.
Although we have not found many target specific regressions (Figure 11), exploring different target architectures and seeing if we find a difference between them would be interesting to study.
Furthermore, developing any kind of higher-level understanding of the regressions DEAD finds could enhance any present analysis done and enable a better distinction of regressions found within one regression commit. The latter use-case would also improve DEADs ability to (automatically) test new changes.
Inter-Optimization level and inter-compiler "regressions" have not been explored in this work. Analyzing such "regressions" coupled with a better higher-level understanding of them could show the strengths and weaknesses of the different projects and potentially allow to pinpoint and transfer insights between them.

ACKNOWLEDGMENTS

First and foremost I want to thank my advisor Theodoros Theodoridis for all the writing tips, code reviews, and meetings that more often than not ended up taking longer than scheduled because of interesting tangents.
I also want to thank Robin Staab, Ben Fiedler and Lucas Brunner for extensive advanced rubber-ducking[6] and feedback on my writing, as well as Oliver Schwarzenbach and Joël Mathys for further feedback on my writing.

REFERENCES

2022 (accessed March 31, 2022). CompCert Website. https://compcert.org.

Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 82–92. https://doi.org/10.1145/3178372.3179521

James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (feb 2020), 36 pages. https://doi.org/10.1145/3363562

Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, and Josep Torrellas. 2018. An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 126 (oct 2018), 29 pages. https://doi.org/10.1145/3276496

Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ trans. syst. LSI des. methodol.* 9, 0 (2016), 21–29.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2013. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14* (Edinburgh, United Kingdom). ACM Press, New York, New York, USA.

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. https://doi.org/10.1145/3428264

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. *Alive2: Bounded Translation Validation for LLVM.* Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: Differential Testing for Embedded Compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala* (Amsterdam, Netherlands) *(SCALA 2016)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/2998392.2998397

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing China). ACM, New York, NY, USA.

Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. *Testing Static Analyses for Precision and Soundness.* Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3368826.3377927

Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne Switzerland). ACM, New York, NY, USA.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *SIGPLAN not.* 46, 6 (June 2011), 283–294.

[6] **Def. Advanced Rubber-Ducking**: Compared to normal rubber-ducking, advanced rubber-ducking requires the rubber duck to be knowledgable about the topic that is rubber-ducked about.

A ADVANTAGE AND PERCENTAGE PLOTS

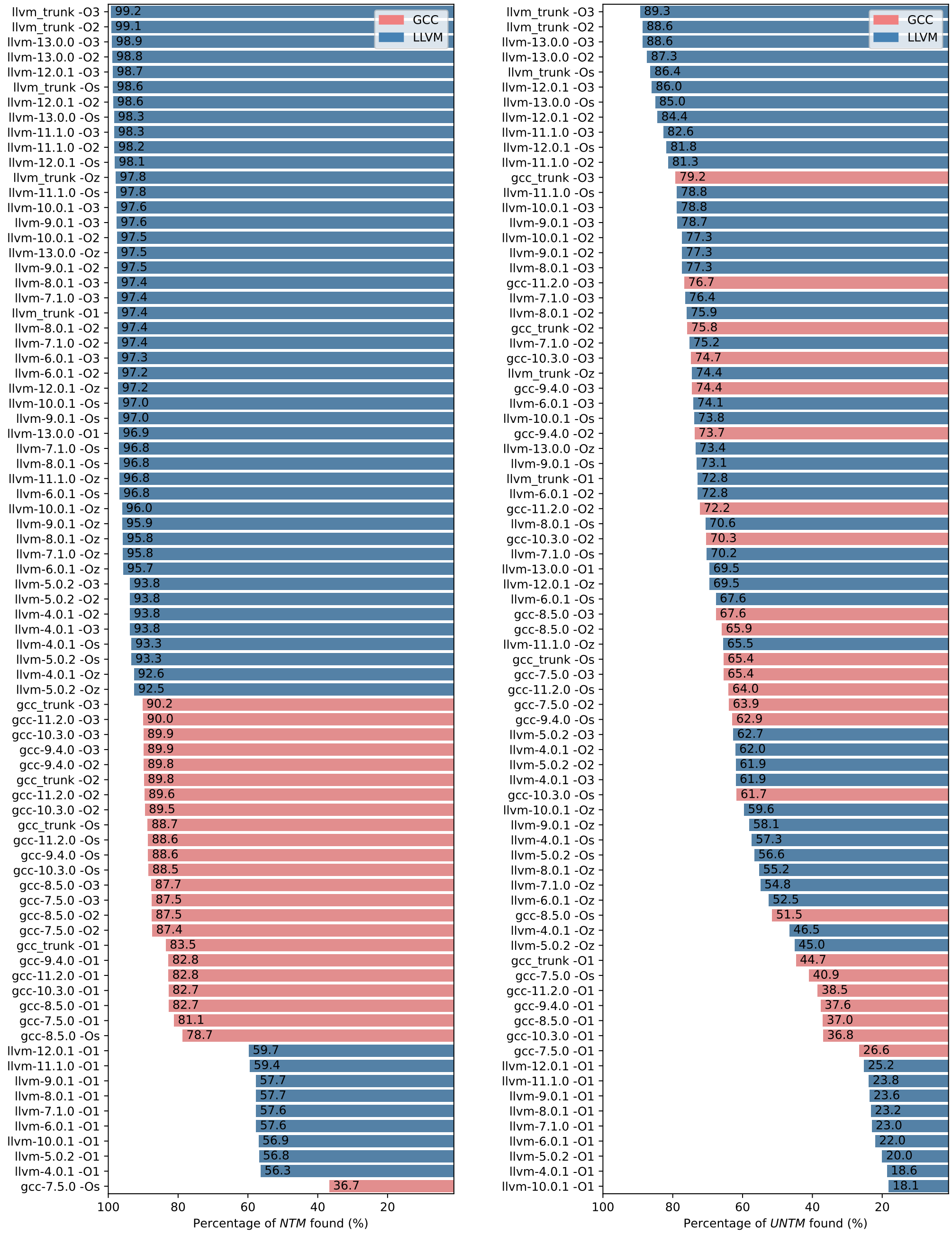


Fig. 15. Comparisons on NTM and $UNTM$ for all optimization levels and major releases.

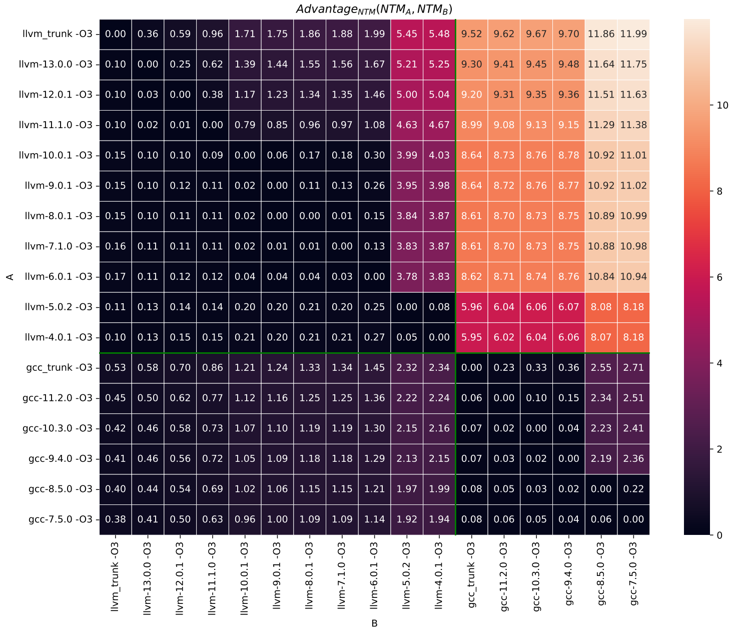


Fig. 16. Advantage plot for NTM and O3

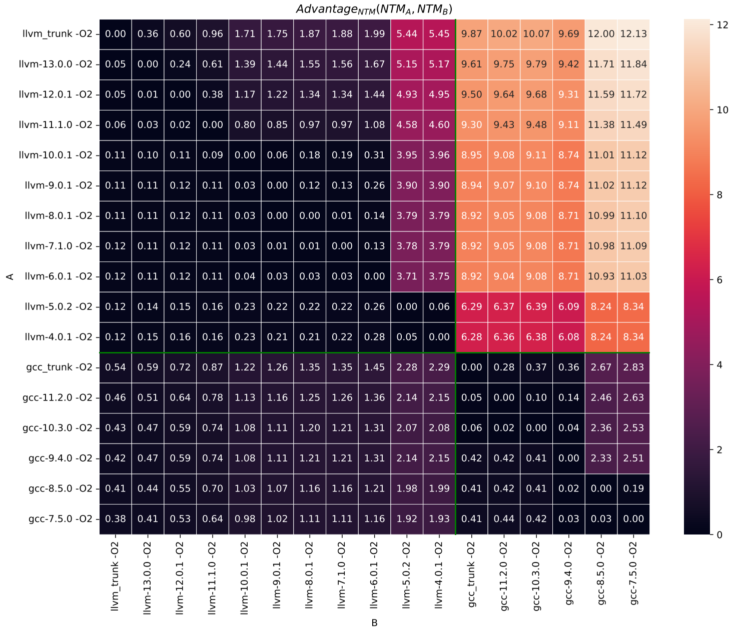


Fig. 17. Advantage plot for NTM and O2

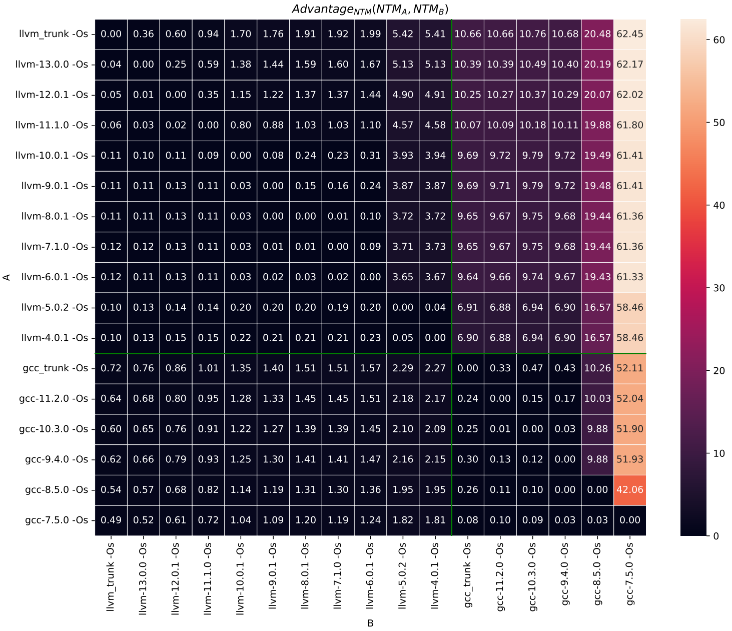


Fig. 18. Advantage plot for NTM and 0s

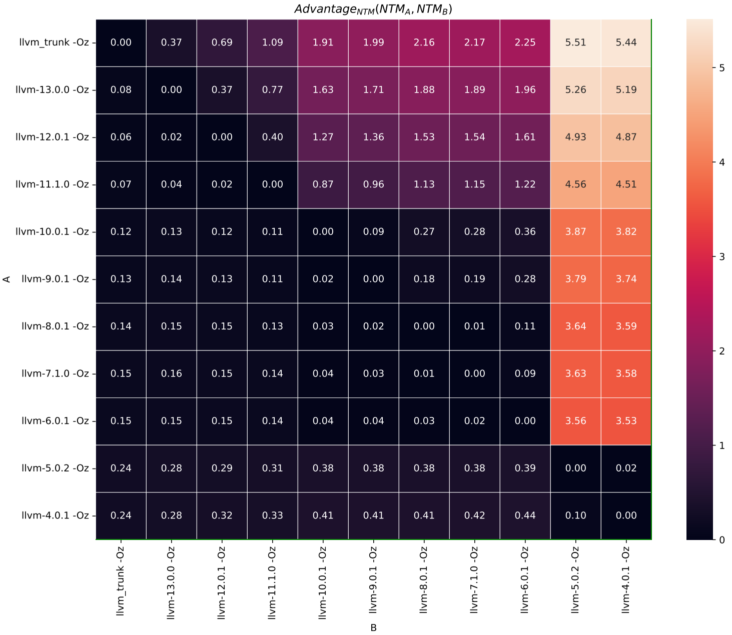


Fig. 19. Advantage plot for NTM and 0z

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

DEAD: Dead Code Elimination based Automatic Differential Testing

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s): Girsberger

First name(s): Yann Willem

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

15.3.2022, Winterthur

Signature(s)

Yann Girsberger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.